empros gmbh
process & information management services

An Introduction to Test Driven Development Using xUnit4Delphi

Sascha Frick empros gmbh

Contents

Introduction

Refactoring can help you to improve the design of existing code, making it easier to understand and maintain. Refactoring is driven by the fundamental idea of evolutionary design and the notion that writing program code is just a design activity, however one that is concerned with a lot of detail. Successful Refactoring requires early and continuous unit testing. XUnit4Delphi - a framework for writing automated unit tests in Delphi - can help you write more robust code by writing automated regression tests. In the following article, I want to introduce you to the idea of Test Driven Development (TDD) using refactoring and unit testing to write clean code that works!

One word of warning before we start! If you have never done TDD before, you might be in for a shock or at least for some surprises. You even might – as some people do - vigilantly or even aggressively reject TDD as silly or downright wrong. But I hope I can wet your appetite and you will – in the words of Erich Gamma – become "test infected" and start writing more tests more often and earlier. And who knows, maybe we will see more articles that use TDD as a straight forward means in communicating new ideas and concepts.

Ok, let's roll up our sleeves and start right off! To get the most out of this journey, I suggest you fire up Delphi and type along as we go. Please note, you need to have xUnit4Delphi installed and properly configured in order to make the examples run.

You can download xUnit4Delphi from
http://www.empros.ch/vielfach/xunit4delphi/xunit4delphi.html.

Clean Code that Works

Here we go: Test Driven Development (TDD) is a simple yet powerful technique to write clean code that works. TDD helps us in this endeavor by providing some simple rules we can follow. As Kent Beck puts it: *"There are many forces that drive us away from clean code, and even from code that works."* So how does TDD help us to achieve clean code that works? Here are two simple rules:

- Write new code only if an automated test fails.
- Rigorously eliminate duplication.

There are some important implications stemming from these rules:

- We must have automated tests to begin with.
- In order to keep a design (we consider code just a very detailed form of design) testable using automated tests, it must consist of highly cohesive but loosely coupled components.
- We therefore must practice an evolutionary design style, where we can get feedback on what we are doing early and often. You might have heard the phrase *"Code a little – test a little!",* which is a big step forward compared to the usual developers habit of *"Code a lot – test a little or not at all"*. With TDD it is just the other way round: *"Test a little – code a little!"*.

The general TDD cycle goes as follows:

1. Write a test.
2. Make it run.
3. Make it right.

Remember: The goal is clean code that works. The usual way to achieve this is to always write clean code and make it gradually work. In TDD it is just the other way round. Again, Kent Beck gives an eloquent summary on this approach: *"First we solve the 'that works' part of the problem. Then we'll solve the 'clean code' part. This is the opposite of architecture-driven development, where you solve 'clean code' first, then scramble around trying to integrate into the design the things you learn as you solve the 'that works' problem."*

A Simple Iterator Example

We will look at a very simple example in order to get a feeling for how TDD works. We want to write an Iterator class we can use to iterate over the elements of `TList` instances. You are probably familiar with the basic idea of an Iterator: it allows you to traverse the elements of an aggregate object, in our example a `TList` object, in different ways without forcing you to bloat the aggregate object's interface with operations for different traversal strategies. Furthermore, using an Iterator several clients can traverse the same list at the same time independently, i.e. without undesired side effects. Our Iterator will be quite simple, offering only two public methods:

```
function HasNext: Boolean;
```

Tests whether there is a next element that can be accessed using `Next`. Return `True` if there is a next element, otherwise `False`.

```
function Next: Pointer;
```

Return the next element in the list.

Using an Iterator, we can write very compact and straightforward code like:

```
while iterator.HasNext do SomethingClever(iterator.Next);
```

A First Test

Before we can use our Iterator in this fashion though, we must create an Iterator object and supply it with the list we want to traverse. So we will start there. We create a new project an save it under the name `IteratorTest`. We change the project source code to look like listing 1. The line `DUnitGUI.CreateGUIRunner;` runs the GUI-based test runner of the xUnit4Delphi framework.

```
program IteratorTest;

uses
  iterTest,
  Forms,
  DUnitGUI,
  iter in 'iter.pas';

{$R *.res}

begin
  Application.Initialize;
  DUnitGUI.CreateGUIRunner;
  Application.Run;
end.
```

Listing 1: Project Source Code

```
unit iterTest;

uses
  // Delphi
  Classes, SysUtils

  // xUnit4Delphi
  ,DunitFramework;

type

  TMgIteratorTest = class(TTestCase)
  private

  end;

initialization
  RegisterTest(TIteratorTest.Suite);
end.
```

Listing 2: iterTest Source Code

Now, let's implement our first test: Create an new unit `iterTest` according to listing 2. Note the initialization code that ensures our Iterator tests will get registered with the framework. Class `TIteratorTest`, which is a subclass of the framework provided class `TTestCase`, will host all of the tests for our yet to be born list Iterator. This class represents what is called the "fixture" for our tests. We will look at this a little later. For each test we simply add a no-argument method in the published section of the class. This allows the framework to automatically collect all the tests at run time. Note that xUnit4Delphi simply collects all published methods regardless of their names and possible argument list. So be very careful to only put methods in the published section that actually represent tests and make sure that you don't specify any arguments for your test methods, since xUnit4Delphi will have no way of telling what argument values to pass to that method when calling it. By convention, all test methods must start with the prefix `Test` followed by a descriptive name that clearly expresses the intention of the test. So, for our first test we add a method named

`TestIteratorWithEmptyList` in the published section of class `TIteratorTest`. The implementation of this method forms our first test which is quite simple, as you can see from listing 3.

```
unit iterTest;

uses
  // Delphi
  Classes, SysUtils

  // xUnit4Delphi
  ,DunitFramework;

type

  TIteratorTest = class(TTestCase)
  published
    procedure TestIteratorWithEmptyList;
  end;

implementation

procedure TIteratorTest.TestIteratorWithEmptyList;
var
  list: TList;
  iter: TListIterator;
begin
  list := TList.Create;
  iter := TListIterator.Create(list);
  try
    AssertTrue('HasNext should be false for empty list',not iter.HasNext);
  finally
    iter.Free;
    list.Free;
  end;
end;

initialization
  RegisterTest(TIteratorTest.Suite);
end.
```

Listing 3: TestIteratorWithEmptyList

First, we setup a new and empty `TList`. Next, we create a new `TListIterator` object for this list. Then, we check that our iterator's `HasNext` method correctly returns false since there are no (more) elements to traverse. Because we want our list and iterator to be freed no matter what, once we are done using them, we wrap their usage with a try/finally block. Unfortunately, this code will not compile for a very simple and hopefully obvious reason: We don't have a `TListIterator` class yet. So let's tackle that one next.

We create a new unit and save it under the name `iter.pas` and add all the code that is needed to make the Delphi compiler happy, see Listing 4. Don't forget to add the `iter` unit to the uses clause of our test unit `iterTest`.

```
unit iter;

interface
uses
  classes;

type
  TListIterator = class
  public
    constructor Create(AList: TList);
    function HasNext: Boolean;
  end;

implementation

{ TListIterator }

constructor TListIterator.Create(AList: TList);
begin

end;

function TListIterator.HasNext: Boolean;
begin

end;

end.
```

Listing 4: iter.pas

Now, let's compile and see what happens. The code compiles without any errors. But Delphi issues a compiler warning telling us that the return value of method `TListIterator.HasNext` could be undefined (If you don't get this compiler warning, make sure that warnings are enabled in the compiler settings). We can fix that by providing a `Result` value in the `HasNext` method. The question is just what value to set the `Result` variable to. Let's return to the TDD cycle of *"write a test, make it run, make it right"* to look for an answer. This cycle implies that we first write a test *and* make sure that it fails! Only *after* the test has failed, we change our code to make the test succeed. In TDD speak this means, we want to get from red to green as fast as possible. But I am getting ahead of myself. So let's first make our test fail by returning `True` in our `HasNext` method: `Result := True;`. After this change, we compile and run our program. If everything is set up properly, you should see the GUI runner of xUnit4Delphi according to figure 1.

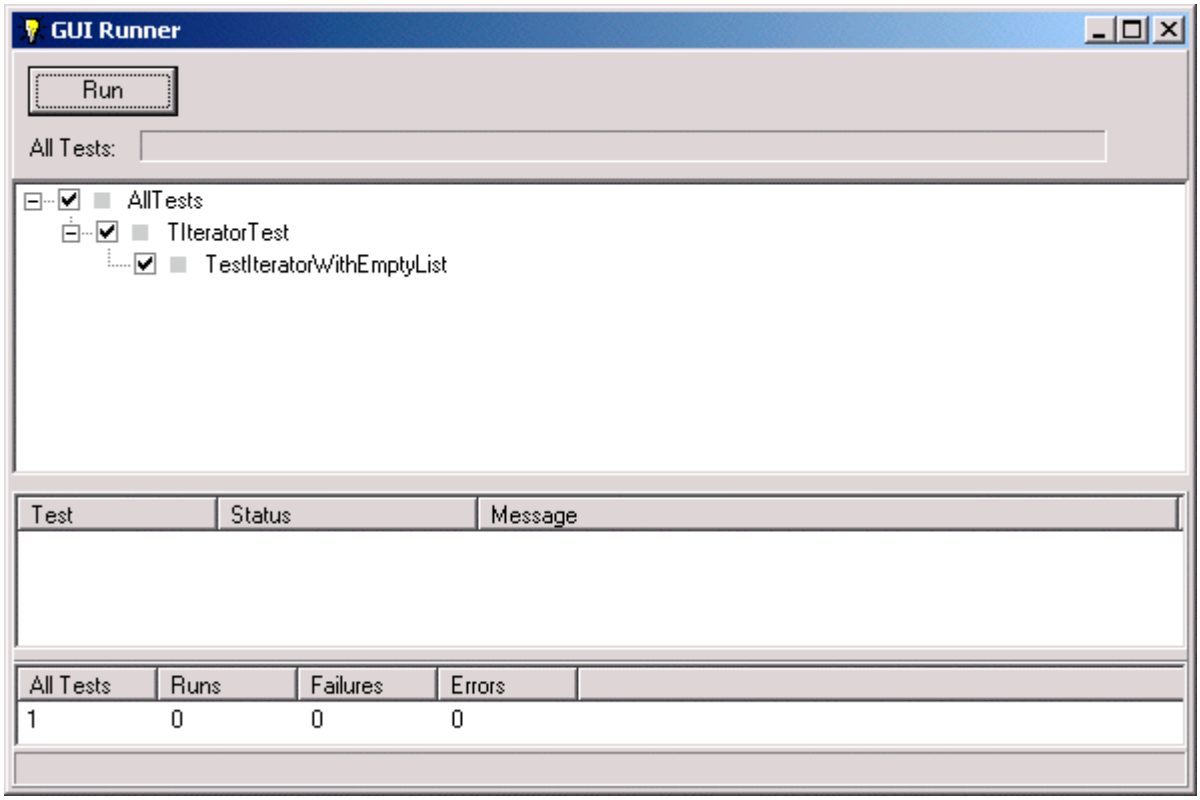

Figure 1: xUnit4Delphi GUI Runner

Hit *F9* to execute our test and see it fail with the failure message we provided in our call to the `AssertTrue` method, see figure 2.

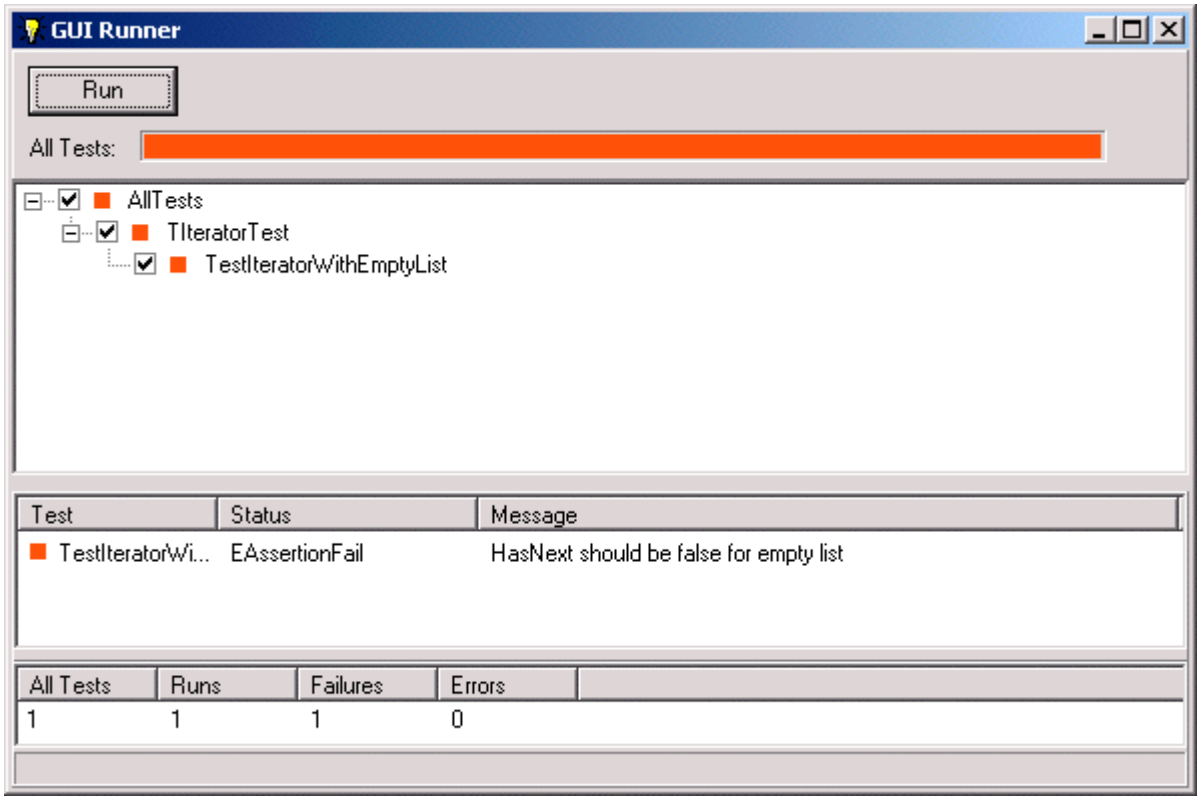

Figure 2: TestIteratorWIthEmptyList failed

Now let's get from red to green as fast as possible, i.e. let's make the test run. We follow the XP-rule of *"do the simplest thing that might possibly work"* and just change our `HasNext` to return `False`: `Result := False;`. Compile, run and execute the test by pressing *F9* twice, et voila: our test succeeds, see figure 3.

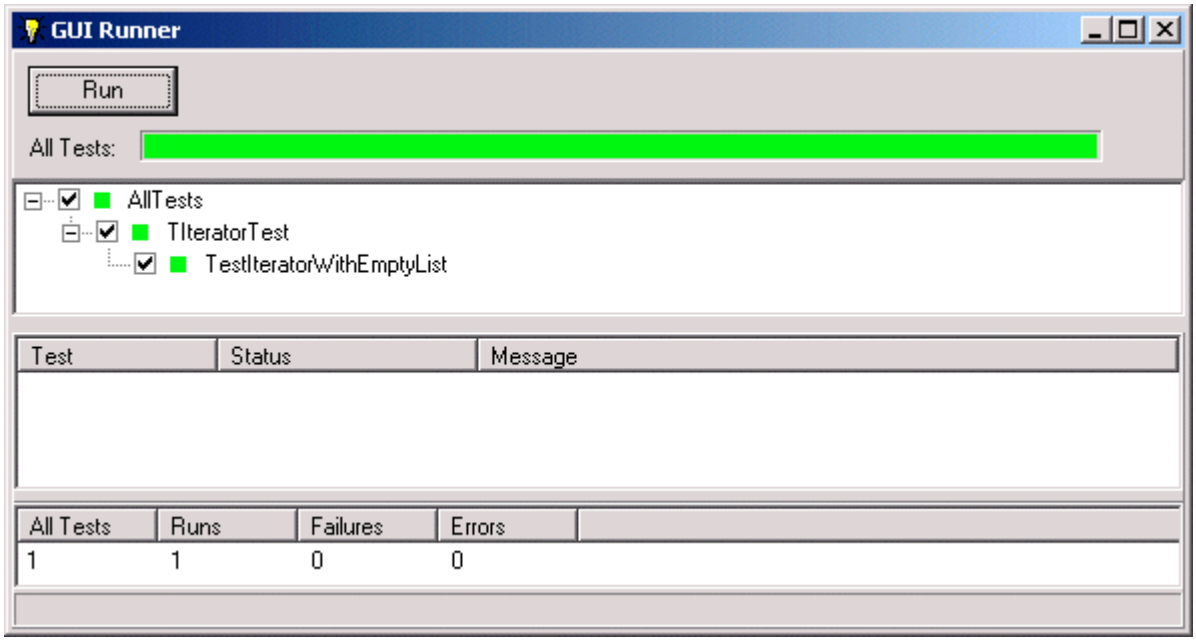

Figure 3: Working TestIteratorWIthEmptyList

At this point you might get the impression that this is a rather childish way to develop software. Returning a constant value like we do in our `HasNext` method is just a cheap trick, our Iterator will never work correctly for a non-empty list. Yes, I know, but we will deal with that problem when it actually becomes a problem. In TDD we take baby steps. Since non-empty lists are the reason we want an Iterator in the first place, we shall address this topic next.

Test Iterator.HasNext

So, let's write a test with a non empty list, say, a list with 3 elements. We expect that calling `HasNext` 3 times will return `True` on each call to signal that there are more elements to iterate over. Thinking in terms of what you expect from objects of a class helps you in designing classes that provide a clear and easy to use interface abstracting away the nitty gritty implementation details. This makes it easy for the user of a class to understand its purpose and how to use it correctly without having to worry about how exactly the class goes about in achieving what it promises to do. This is one reason why TDD helps you with the *"clean code"* part of the *"clean code that works"* mantra.

The implementation of the test looks like listing 5. We first set up a new `TList`, add 3 elements to it and setup a new Iterator using that list. In order to test our Iterator's correct behavior, we will use two local integer variables named `expectedCount` and `actualCount` to hold the expected and the actual element count respectively, following the xUnit naming conventions for actual and expected test values. With

these two variables, writing our actual test expression is rather simple: `AssertEquals(expectedCount,actualCount);`. Of course we need to set the values of `actualCount` and `expectedCount` before we perform this check. Therefore, we set the `expectedCount` as follows: `expectedCount := list.Count;`, since we expect that we can call `HasNext` on our Iterator `list.Count` number of times. This implies that we will need to call `HasNext` for as long as it returns `True` and increment `actualCount` on each call: `while iter.HasNext do Inc(actualCount);`. We are then ready to perform the actual check as shown above. The rest of the method is comprised of the necessary clean up code to free the Iterator and dispose of the `TList` object and its elements.

```
procedure TIteratorTest.TestIteratorHasNextForNonEmptyList;
var
  list: TList;
  iter: TListIterator;
  i: Integer;
  actualCount: Integer;
  expectedCount: Integer;
begin
  list := TList.Create;
  for i := 0 to 2 do list.Add(Pointer(i));
  iter := TListIterator.Create(list);
  try
    actualCount := 0;
    expectedCount := list.Count;
    while iter.HasNext do Inc(actualCount);
    AssertEquals(expectedCount, actualCount);
  finally
    iter.Free;
    list.Free;
  end;
end;
```

Listing 5: TestIteratorHasNextForNonEmptyList failed

The code we just produced compiles just fine. The question now is, what will happen if we run it. Before you hit *F9* to find out, pause for a moment and think about the expected outcome. We have two tests so far: `TestIteratorWithEmptyList` and `TestIteratorHasNextForNonEmptyList`. It is safe to assume, that the first test will still run and being aware of our implementation of class `TListIterator` and our expectations stipulated in the second test, we conclude that the latter will fail. Thinking beforehand about the expected outcome– though obvious for this very simple example – is a very important aspect of TDD, since it helps us stay focused on what our code should be able to do. We hit *F9* twice to find our hypothesis verified, see figure 4.

Ok, the second test fails as expected. Now, we need to remedy that situation by providing a more sophisticated implementation of our Iterator that actually does iterate the list's element, see listing 6 for an implementation that gets us from red to green. We compile and run the program and get rewarded with a green bar. Note that even though we changed the implementation of the `HasNext` method, our initial `TestIteratorWithEmptyList` test still works. This is very important because it shows we have not broken existing functionality when we altered the implementation

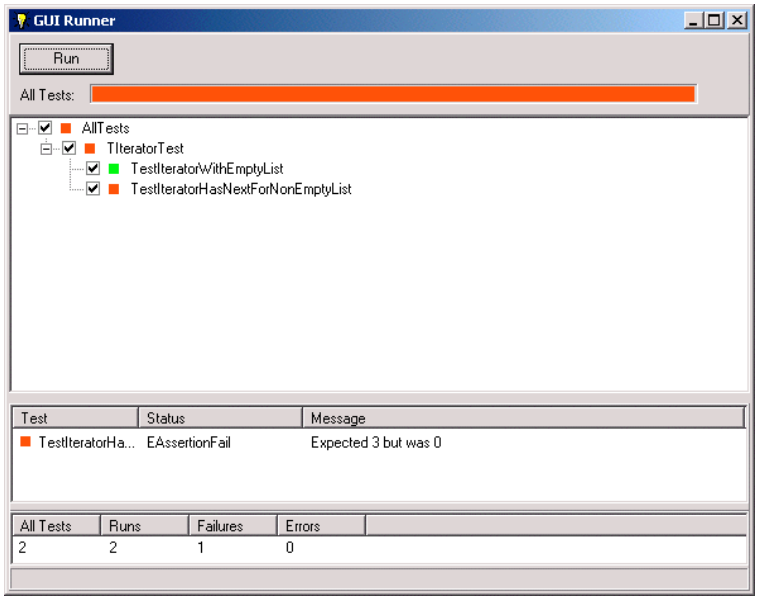

Figure 4: TestIteratorHasNextForNonEmptyList failed

of our Iterator. Being able to easily do regression testing in this fashion is one of the many benefits of TDD and a key factor to successful and safe refactoring.

```
unit iter;

interface
uses
  classes;

type
  TListIterator = class
  private
    FCursor: Integer;
    FList: TList;
  public
    constructor Create(AList: TList);
    function HasNext: Boolean;
  end;

implementation

constructor TListIterator.Create(AList: TList);
begin
  FList := AList;
  FCursor := 0;
end;

function TListIterator.HasNext: Boolean;
begin
  Result := FCursor < FList.Count;
  if Result then Inc(FCursor);
end;

end.
```

Listing 6: TListIterator implementation

Get Rid of Stinking Code: Refactor

We still have to provide the Next method that allows users of our class to get the current element from the list. But before we start writing a new test to exercise that behavior on our Iterator, let's pay a visit to our test code. Uh oh, can you smell it? Though faint, a wary programmer's nose can easily detect it, the smell of duplication! Have a look at our two test methods. Besides the actual test code, they both contain code that creates `TList` and `TListIterator` objects along with the appropriate clean-up code. Remember one of the TDD rules stated earlier: *"Rigorously eliminate duplication"*. This rule not only applies to our `TListIterator` but to our test code also. Duplication is evil and one of the root causes for software rot. So we are in for a refactoring to get rid of this duplication. Since both tests exercise the same class, it seems natural that they both have similar setup code to provide a fixture for the actual test code to run. As mentioned before, it is the test class that provides this fixture. So it seems correct to expect xUnit4Delphi to provide a common way to setup the fixture. This common way comes as a method called `SetUp` which is guaranteed to be called by the framework for each test just before it gets executed, i.e. before a test method is called. So we can move the `TList` and `TListIterator` creation code to the setup method, therefore eliminating the duplicate creation code. In order for our test methods to be able to use these objects, both the list object and the Iterator get a promotion from mere local variables to member variables. We then have to change all occurrences of the local variables with the member variables; with a little help from the Delphi DIE this is a simple matter of replacing `list` with `FList` and `iter` with `FIterator` respectively. Fine, you say, the `Setup` method is all nice and good to create our objects, but what about the code that frees the Iterator and the list? We still find this code in both our test methods. Besides the duplication code smell we wanted to get rid of, we now also have a very ugly form of asymmetry that doesn't smell too well either! We can do better than that! That is why `Setup` has its counterpart in xUnit4Delphi: `TearDown`. This method gets called automatically after your test has been executed. It is the place to put any clean up code to – as the name implies - tear down the fixture. So we can move the destroy code for our list and iterator there. See listing 7 for our refactored solution. Please take note that `Setup` and `TearDown` are protected and virtual. The latter requires the use of override for these methods in our `TIteratorTest` class.

```
unit iterTest;

interface
uses
  // Delphi
  Classes, SysUtils

  // xUnit4Delphi
  ,DUnitFramework,

  iter;

type
```

continued on next page ⇗

continued from previous page

```
TIteratorTest = class(TTestCase)
  private
    FList: TList;
    FIterator: TListIterator;
  protected
    procedure Setup; override;
    procedure TearDown; override;
  published
    procedure TestIteratorWithEmptyList;
    procedure TestIteratorHasNextForNonEmptyList;
  end;

implementation

procedure TIteratorTest.Setup;
begin
  FList := TList.Create;
  FIterator := TListIterator.Create(FList);
end;

procedure TIteratorTest.TearDown;
begin
  FList.Free;
  FIterator.Free;
end;

procedure TIteratorTest.TestIteratorHasNextForNonEmptyList;
var
  i: Integer;
  actualCount: Integer;
  expectedCount: Integer;
begin
  for i := 0 to 2 do list.Add(Pointer(i));
  actualCount := 0;
  expectedCount := FList.Count;
  while FIterator.HasNext do Inc(actualCount);
  AssertEquals(expectedCount, actualCount);
end;

procedure TIteratorTest.TestIteratorWithEmptyList;
begin
  AssetTrue('HasNext should be false for empty list',not
FIterator.HasNext);
end;

initialization
  RegisterTest(TIteratorTest.Suite);
end.
```

Listing 7: Refactored TiteratorTest

Test Iterator.HasNext – Second Part

Now that we have cleaned up our test code, let's go back to our Iterator and the pending implementation of `Next`. Again, we start by writing a test to exercise the desired behavior of our `TListIterator`. We call this test `TestIterate` and

implement it according to listing 8. Of course, this code will not compile, because our Iterator has no `Next` method yet.

```
procedure TIteratorTest.TestIterate;
var
  i: Integer;
  actual: Integer;
  expected: Integer;
begin
  for i := 1 to 3 do FList.Add(Pointer(i));
  i := 0;
  while FIterator.HasNext do begin
    expected := Integer(FList[i]);
    actual := Integer(FIterator.Next);
    AssertEquals(expected ,actual);
    Inc(i);
  end;
end;
```

Listing 8: TIteratorTest.TestIterate

So let's add the Next method and provide an implementation as shown in listing 9 that will cause our test to fail. Compile and run the program to see this verified.

```
function TListIterator.Next: Pointer;
begin
  Result := nil;
end;
```

Listing 9: TListIterator.Next implementation that will cause TestIterate to fail

So, we are now ready to implement the `Next` method correctly. How about the code given in listing 10?

```
function TListIterator.Next: Pointer;
begin
  Result := FList[FCursor];
end;
```

Listing 10: A Next implementation that will not work

Due to the way we implemented the `HasNext` method, the code in listing 10 will not work. You can easily test this: Compile and run the program implementing the `Next` method according to listing 10. As you can see, `TestIterate` still fails, now complaining that the expected value is `<1>`, but the actual value detected was `<2>`. Our Iterator implementation seems to be flawed. So what changes are needed to make our Iterator work as intended? The test failure points us into the right direction. The fact that the expected return value is `<1>` when the actual value returned is `<2>`, suggests an off by one error. I.e. on the first iteration we receive the second list element instead of the first, failing our test. We can easily fix that if we implement `Next` according to listing 11.

```
function TListIterator.Next: Pointer;
begin
  Result := FList[FCursor-1];
end;
```

Listing 11: A working Next implementation

Before we continue, let's have a look at `TestIterate` and `TestIteratorHasNextForNonEmptyList`. We can easily integrate the latter into to former as shown in listing 12, therefore eliminating duplication again.

```
interface
uses
  // Delphi
  Classes, SysUtils

  // xUnit4Delphi
  ,DUnitFramework,

  iter;

type

  TIteratorTest = class(TTestCase)
  private
    FList: TList;
    FIterator: TListIterator;
  protected
    procedure Setup; override;
    procedure TearDown; override;
  published
    procedure TestIteratorWithEmptyList;
    procedure TestIterate;
  end;

implementation

procedure TIteratorTest.Setup;
begin
  FList := TList.Create;
  FIterator := TListIterator.Create(FList);
end;

procedure TIteratorTest.TearDown;
begin
  FList.Free;
  FIterator.Free;
end;

procedure TIteratorTest.TestIterate;
var
  i: Integer;
  actual: Integer;
  expected: Integer;
begin
  for i := 1 to 3 do FList.Add(Pointer(i));
  i := 0;
```

continued on next page ⇗

continued from previous page

```
  while FIterator.HasNext do begin
    expected := Integer(FList[i]);
    actual := Integer(FIterator.Next);
    AssertEquals(expected, actual);
    Inc(i);
  end;
  AssertEquals('List size and number of iterations do not match',
FList.Count,i,);
end;

procedure TIteratorTest.TestIteratorWithEmptyList;
begin
  AssertTrue('HasNext should be false for empty list', not
FIterator.HasNext);
end;

initialization
  RegisterTest(TIteratorTest.Suite);
end.
```

Listing 12: Yet another test code refactoring

There is still some work to do. First, we have to think about the situation where `Next` is called without a preceding `HasNext`. There are two questions to answer. First, what should happen when our Iterator is used in that manner? And second, what will actually happen based on our current implementation? Let's answer the first question first: According to the method description of `Next`, we expect the next element in the list to be returned. If we call `Next` on an Iterator the first time without a preceding `HasNext`, we should get the *first* element in the list, on the second call we should get the second element and so forth. Armed with that information, let's write a test to answer the second question. We call this test `TestNextWithoutHasNext` and implement it like in listing 13.

```
procedure TIteratorTest.TestNextWithoutHasNext;
var
  i: Integer;
  actual: Integer;
  expected: Integer;
begin
  for i := 1 to 3 do FList.Add(Pointer(i));
  expected := 1;
  actual:= Integer(FIterator.Next);
  AssertEquals(expected, actual);
end;
```

Listing 13: TestNextWithoutHasNext

Being aware of how we implemented our `Next` method, we know already that this test will fail or more precisely will generate an `EListError`. Compile and run the program to see this confirmed. As expected, we get an `EListError` because `Next` tried to get an item at list index –1 (`FCursor := 0, 0-1 = -1`). We can easily fix that problem by reworking our Iterator code according to listing 14. But before we do so, let's briefly consider what we have gained by providing a test that proves

something quite obvious. For one thing, we have *confirmed* our assumption that `Next` does not work properly. But the same test will also confirm the correct behavior, once we have fixed the problem. And to make sure, our changes will not have inadvertently broken existing functionality, we have two other tests in place. Again, this is one of the big advantages of TDD!

```
constructor TListIterator.Create(AList: TList);
begin
  FList := AList;
  FCursor := 0;
end;

function TListIterator.HasNext: Boolean;
begin
  Result := FCursor < FList.Count;
end;

function TListIterator.Next: Pointer;
begin
  Result := nil;
  if HasNext then begin
    Result := FList[FCursor];
    Inc(FCursor);
  end;
end;
```

Listing 14: A Refactored TListIterator solution

As for the changes we made: We just moved the iteration code from the `HasNext` method to `Next`. This makes `HasNext` a simple query method that does not alter the state of the Iterator. Wen can now use `HasNext` from within `Next` to make sure, we do not iterate over the end of the list. Compile and run the program. All of our tests run and we are back to green. Since both `TestIterate` and `TestNextWithoutHasNext` use the same code that fills the list, we can move this code into its own method called `FillList` according to listing 15.

```
procedure TIteratorTest.FillList;
var
  i: Integer;
begin
  for i := 1 to 3 do FList.Add(Pointer(i));
end;

...


procedure TIteratorTest.TestIterate;
var
  i: Integer;
  actual: Integer;
  expected: Integer;
begin
  FillList;
  ...
end;
```

continued on next page ⇗

```
                                                    continued from previous page

procedure TIteratorTest.TestNextWithoutHasNext;
var
  actual: Integer;
  expected: Integer;
begin
  FillList;
  ...
end;
```

Listing 15: TiteratorTest using FillList method

Loose Ends

Our Iterator is still not perfect. For one thing the `Next` method can return `nil` when the end of the list is reached. While this is not a problem when `Next` is used within a `do while HasNext` loop, it is still a possible source of trouble. We can argue that calling `Next` on the Iterator when there is no next element, is illegal and – in terms of *Design By Contract* – a breech of contract by the caller. So it is probably best to signal this situation by raising an exception, e.g. an `IllegalStateException`. The Iterator's constructor is another source of trouble. What happens if a user passes `nil` instead of a `TList` instance? The constructor does not account for this and it won't be until the first call to `HasNext` the problem will surface with an ugly Access Violation. This sort of delayed feedback for something that was done wrong earlier is a source for serious trouble and should be avoided as much as possible. Again, it is best to raise an exception – preferably an `IllegalArgumentException` - in the constructor if a `nil` argument is detected. I leave the resolution of these issues as an exercise to the reader.

Closing Remarks on TDD

At first TDD seems slow and just not very efficient. After all, you end up writing a lot of test code, don't you. And to make things worse, every time you refactor your code the tests will probably have to be changed as well. So this can't possibly work for any real world project, right? Wrong! If you only ran your tests once or twice during development, then writing automated tests would not be a very economic thing to do. But remember: We want to run our tests regularly, basically quite as often as we hit the compile button.

When first confronted with the ideas that lie at the heart of TDD, people often react heftily, sometimes even aggressively. As a consultant and trainer I am asked quite often, whether I really believe that TDD is the right way to develop software. Personally I don't think that these people are asking the right question. For me, it is not a matter of whether TDD is the right way to develop software but rather whether TDD is an adequate approach in helping us to write working software that is easy to understand and therefore easy to change. TDD is no panacea and there are situations it might be inappropriate. I have been using TDD for quite some time now with overwhelming success. Never before did I feel so much in charge of what I do, never before was I so confident to reconsider, refine and change existing code without fear of breaking existing functionality. Knowing my own deficiencies, I cannot

overestimate the power that TDD offers me in developing working software on budget and in time.

Sascha Frick *is founder and owner of empros gmbh. He works as an independent Coach and Trainer for Object Orientation. He seems to hear himself repeating the question "Do you have a test to prove your assumption?" all too often during his work. He can be reached via email at vielfach@empros.ch.*